

AD-A133 264

CLASSIFYING BUGS IS A TRICKY BUSINESS(U) YALE UNIV NEW
HAVEN CT DEPT OF COMPUTER SCIENCE W L JOHNSON ET AL,
AUG 83 YALEU/CSD/RR-284 N00014-82-K-0714

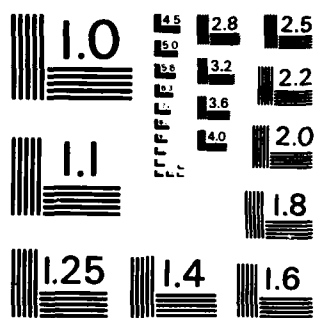
1/1

UNCLASSIFIED

F/G 9/2

NL

END
DATE
FILMED
NO
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A133264



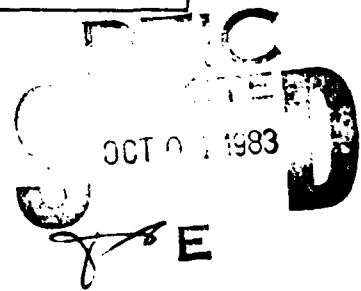
CLASSIFYING BUGS IS A TRICKY BUSINESS

W. Lewis Johnson, Stephen Draper
and Elliot Soloway

YaleU/CSD/RR #284

August 1983

DTIC FILE COPY



YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

This document has been approved
for publication and sale; its
distribution is unlimited.

83 10 04 092

CLASSIFYING BUGS IS A TRICKY BUSINESS

W. Lewis Johnson, Stephen Draper
and Elliot Soloway

YaleU/CSD/RR #284

August 1983

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER #284	2. GOVT ACCESSION NO. AD-A133264	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Classifying Bugs is a Tricky Business		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) W. Lewis Johnson, Stephen Draper and Elliot Soloway		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0714
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Yale University New Haven, CT 06520		10. PROGRAM ELEMENT PROJECT, TASK AREA & WORK UNIT NUMBERS NR 154-492
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research (Code 458) Arlington, VA 22217		12. REPORT DATE August 1983
		13. NUMBER OF PAGES 17
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION, DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES 7th Annual NASA/Goddard Workshop on Software Engineering, Baltimore, 1982.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Software Engineering Program Bugs Program Understanding Programming Plans		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) see attached page		

DD FORM 1473
1 JAN 73EDITION OF 1 NOV 65 IS OBSOLETE
S N 0102- LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

In order to build a computer-based programming tutor for novice programmers, we needed to first classify the bugs we found in their programs on the basis of type and frequency. However, the enterprise of classification turns out to be a complicated process. While one may want to be able to simply use features in the program itself as the basis for the classification, it turns out that such a scheme will result in classifications that seem to miss the mark, i.e., the classifications will not tell you what misconception the programmer was operating under which caused the bug. To remedy this situation, we argue that the programming plans that the programmer intended to use should be the basis for a classification scheme. Thus, a bug classification must take the programmer directly into account. In this paper, we compare several different methods of bug classification currently being used in software engineering projects, and show their weaknesses; while our method of using intended programming plans is not without problems, we argue that it presents a better alternative than the other methods currently being employed.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
EVAL	
Distribution/	
Accession Codes	
/or	
DTIC	
A	

To Appear: 7th Annual NASA/Goddard Workshop on Software Engineering, Baltimore, 1982.

Classifying Bugs is a Tricky Business

W. Lewis Johnson *

Stephen Draper **

Elliot Soloway *

* Department of Computer Science

Yale University

P.O. Box 2158

New Haven, Connecticut 06520

** Institute for Cognitive Science

University of California, San Diego Mail Code C015

La Jolla, California

This work was co-sponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, Contract No. N00014-82-K-0714, Contract Authority Identification Number, Nr 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

1. Context: Motivation and Goals

About 2 years ago we decided to build a computer-based programming tutor to help students learn to program in Pascal; we wanted the system to identify the *non-syntactic* bugs in a student's program and tutor the student with respect to the misconceptions that might have given rise to the bugs. The emphasis was on the system understanding what the student did and did not understand; we felt that simply telling the student that there was a bug in line 14 was not sufficient --- since oftentimes the bug in line 14 was really caused by a whole series of conceptual errors that could not be localized to a specific line in the program. However, in order to design the system we needed to know what bugs students did make in their programs and what misconceptions they typically labored under. On the basis of bug types found in a number of pencil-and-paper studies with student programmers (novices, intermediates, and advanced) [9, 10], we built and classroom tested a first version of such a programming tutor [11]. In the process of testing that system we instrumented the operating system on a CYBER 175 to automatically collect a copy of each syntactically correct program the student programmers attempted to execute while sitting at the terminal; we call this form of data "on-line protocols". We collected such protocols on 204 students for an entire semester (7 programming assignments). We have systematically analyzed only a small portion of these data: the basis for this paper is the hand analysis of the first syntactically correct program that students generated for their first looping assignment,¹ i.e., 204 programs.

The story we tell in this paper deals with our experiences in analyzing these 204 on-line protocols. In particular, we will describe the observations we made in trying to build a bug classification scheme; the actual details of what bugs we found, their frequency, etc. can be found in [5]. The key observation is the following: while one might think that building a classification scheme for the bugs would be straightforward, it turns out not to be so simple; in fact, we will argue that:

Bugs cannot be uniquely described on the basis of features of the buggy program alone; one must also take the programmer's intentions and knowledge state into account.

2. A Simplified Example

Consider the problem statement in Figure 1, which is a simplified version of the first looping problem that the students in our study had to solve in Pascal. From a novice's perspective the difficult part of this problem is making sure that the negative inputs are filtered out before they are processed. There are two common approaches to solving this type of problem in an Algol-like language such as Pascal. In Figure 2 we depict a solution in which a negative input causes

¹This problem is given in Figure 8, which will be discussed in section 4.

execution of one branch of a conditional, while a non-negative input causes execution of the major computation of the loop. We call this type of structure a *Skip-guard Plan*:² a conditional statement is used to guard the main computation from illegal values. Notice that one pass through the loop will be made for each input value. The second approach is given in Figure 3; here an embedded loop filters out the illegal values. Notice that one pass through the outside loop will be made for each — and only each — legal value. We call the nested loop structure an *Embedded Filter Loop Plan*.

Write a program that reads in integers, that represent the daily rainfall in the New Haven area, and computes the average daily rainfall for the input values. If the input is a negative number, do not count this value in the average, and prompt the user to input another, legal value. Stop reading when 99999 is input; this is a sentinel value and should not be used in the average calculation.

Figure 1: Simplified Looping Problem

```

.....
.....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
  BEGIN
    IF RAINFALL < 0
    THEN
      WRITELN('BAD INPUT, TRY AGAIN')
    ELSE
      BEGIN
        TOTAL := TOTAL + RAINFALL;
        DAYS := DAYS + 1;
      END;
    READ(RAINFALL);
  END;
END;
.....
.....

```

Figure 2: Using a *Skip-Guard Plan*

Now consider the buggy program in Figure 4. The problem with this program is that if the user first types a negative input, and then types the sentinel value 99999, this value will — incorrectly — be processed as a legitimate value. A number of questions come to mind:

1. How should we classify this bug?
2. What piece of code is to blame?
3. What mental error on the student's part might have caused this bug?

²See [8, 3, 9] for a more complete discussion of programming plans.

```

.....
.....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
  BEGIN
    WHILE RAINFALL < 0 DO
      BEGIN
        WRITELN('BAD INPUT, TRY AGAIN');
        READ(RAINFALL)
      END;
    IF RAINFALL <> 99999 THEN
      BEGIN
        TOTAL := TOTAL + RAINFALL;
        DAYS := DAYS + 1;
        READ(RAINFALL)
      END;
    END;
  END;
.....
.....

```

Figure 3: Using an *Embedded Filter Loop Plan*

4. What piece of code should we change to make the program correct?

In order to answer these questions, however, we need to answer another one first:

What programming approach was the user trying to implement? That is, did the student intend to implement the *skip-guard plan* or did he try to implement the *embedded filter loop plan*?

Answers to the first 4 questions will be different depending on how we answer this last question.

```

.....
.....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
  BEGIN
    WHILE RAINFALL < 0 DO
      BEGIN
        WRITELN('BAD INPUT, TRY AGAIN');
        READ(RAINFALL)
      END;
    TOTAL := TOTAL + RAINFALL;
    DAYS := DAYS + 1;
    READ(RAINFALL)
  END;
.....
.....

```

Figure 4: Sample Buggy Program

We will continue this example by presenting first an argument that supports the choice of the *skip-guard plan*, and then an argument that supports the choice of the *embedded filter loop plan*; we will then describe a basis for making a choice between the two competing

positions. Consider, then, Figure 5 in which we depict the buggy program again, plus a generalized, template version of the *skip-guard plan*. We can describe the buggy program in terms of a difference description between the it and the generalized plan. As shown in Figure 5, there are 3 differences:

1. need an IF instead of a WHILE inside the loop,
2. have an extra read inside the loop,
3. will always execute the processing steps since there is no way to skip around the processing.

The first difference is a plausible bug for a novice to make; in our examination of novice programs we have seen novices confuse IF and WHILE: students sometimes construct a loop with simply an IF, and sometimes they use just the test part of the WHILE statement³ [2, 6]. Similarly, the second difference is also plausible for novices; again, we have found that novices often add bits of spurious code, oftentimes attempting to mimic the redundancy they often use in formulating plans and actions in the real world. Finally, if we assume that the programmer really meant to simply test RAINFALL, then all that is missing is an ELSE to cause the skip around the computation; novices notoriously have trouble with the ELSE parts of conditionals. Thus, the buggy code in Figure 5 is not that different from the *skip-guard plan*; when considering differences from only this plan it is entirely conceivable that the novice programmer was trying to implement this plan in his code.

Now consider Figure 6 in which we again depict the buggy program. This time, however, we show differences between it and a generalized, template version of an *embedded filter loop plan*. Notice that the code matches the plan well; the only bug is a missing guard before the code that processes the input: the running total update and the counter update must be protected from including a sentinel value in the computation.

The analysis in Figures 5 and 6 would lead to different answers to the first 4 questions above. For example, if we believe that the analysis in Figure 5 is correct, we might say the following to the student:⁴

It seems that you are having some trouble with conditional statements. For example, did you realize that there exists a statement called IF that allows you to test

To correct your program, you might want to add an ELSE clause...

³While this may seem strange to us as expert programmers, if we take a moment to reflect, we can see that using WHILE for a conditional and a loop, and IF for only the conditional part is somewhat arbitrary, given their meanings in English.

⁴We do not want to argue about the best pedagogical strategy for interacting with the student; that in itself is a very difficult question. The particular response shown is simply meant to illustrate one type of response to this situation.

<pre> READ(RAINFALL) WHILE RAINFALL <> 99999 DO BEGIN WHILE RAINFALL < 0 DO BEGIN WRITELN('BAD INPUT, TRY AGAIN'); READ(RAINFALL) END; TOTAL := TOTAL + RAINFALL; DAYS := DAYS + 1; READ(RAINFALL) END; END; </pre>	<p>Skip-Guard Plan</p> <pre> IF x < min THEN BEGIN print error message END ELSE BEGIN process input END </pre>
--	--

BUG DESCRIPTION:

1. need an IF instead of a WHILE
2. have an extra READ in inner loop
3. missing ELSE; processing of input will never be skipped

Figure 5: Bug Description Assuming *Skip-Guard Plan*

Moreover, we would classify the bugs as an (1) incorrect statement type, (2) spurious read, (3) missing ELSE. On the other hand, if we believe that the analysis in Figure 6 is correct, then we might say something like the following to the student:

You should notice if the sentinel value follows the input of a negative value that your program will compute an incorrect average.

The bug type then might be a missing guard (conditional) plan.

By this time the reader's intuition is surely saying that the correct analysis of the buggy program in Figure 4 is that the programmer intended to implement an *embedded filter loop plan*. The bug counts (3 for the *skip-guard plan* and 1 for the *embedded filter loop plan*) provide quantitative support for this decision. However, we feel that the key in the decision process --- and the basis for our intuition --- is our *understanding* of the student's program provided by the plan analysis in Figure 5: thus, the bug categorization and bug count follow from our understanding of the program --- and not the other way around. We purposely choose an example over which there would be little controversy. However, the point was (1) to show how much reasoning we often do about programs implicitly, and (2) to show how different bug categorization and bug counts could be as a function of choice of intended underlying plan.

While the above decision was relatively clear, let us perturb the buggy code a bit further and

```

....
....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
  BEGIN
    WHILE RAINFALL < 0 DO
      BEGIN
        WRITELN('BAD INPUT, TRY AGAIN');
        READ(RAINFALL)
      END;
    TOTAL := TOTAL + RAINFALL;
    DAYS := DAYS + 1;
    READ(RAINFALL)
  END;

```

Embedded Filter Loop Plan

```

WHILE x < min DO
  BEGIN
    print error message
    READ x
  END
sentinel guard plan
process input

```

BUG DESCRIPTION:

1. missing conditional (guard) on processing the input

Figure 6: Bug Description Assuming *Embedded Filter Loop Plan*

see how murky these type of decisions can -- and do -- become. In Figure 7 we show three buggy program fragments; let us compare the bug categorization and bug counts using the two alternative plans for each of the programs.

• **Figure 7a**

- Using the *embedded filter loop plan* we get the following bug differences:
 1. the WHILE and IF keywords have been interchanged
 2. there is a missing read for a new value
 3. there is a missing guard on the subsequent input processing
- Using the *skip-guard plan* we get the following bug differences:
 1. missing ELSE on the internal IF

• **Figure 7b**

- Using the *embedded filter loop plan* we get the following bug differences:
 1. the WHILE and IF keywords have been interchanged
 2. there is a missing guard on the subsequent input processing
- Using the *skip-guard plan* we get the following bug differences:
 1. spurious READ
 2. missing ELSE on the internal IF

• Figure 7c

- Using the *embedded filter loop plan* we get the following bug differences:
 1. missing read for a new value
 2. there is a missing guard on the subsequent input processing
- Using the *skip-guard plan* we get the following bug differences:
 1. the WHILE and IF keywords have been interchanged
 2. missing ELSE on the internal IF

We would argue that the programmer of the code in Figure 7a intended to encode a *skip-guard plan*: again, the bug counts (3 for the *embedded filter loop plan* and 1 for the *skip-guard plan*) support the intuition that it is more plausible that the programmer simply left out an ELSE, as opposed to swapping keywords, etc. However, the code in Figures 7b and c are not so easily analyzed: the bug counts are the same and the plausibility of the bug types are reasonably similar. In order to make a reasoned decision we need to bring *other* evidence from the program to bear. For example, in Figure 7b the programmer used a WHILE loop to correctly implement the outer loop; this is some evidence that he understands how and when to use this construct. Thus, we might be confident that the programmer really meant IF in the program in Figure 7b. On the other hand, the inclusion of the spurious READ is unsettling. However, the program in Figure 7c is certainly the most problematic: the bug counts are the same, the plausibility of the bugs are similar, and the additional outside information is equivocal. The moral of this program is that it can be exceedingly difficult to make decisions about plans --- and bugs --- by *simply looking at the code*.

The point of these latter examples is to illustrate how quickly the decision about what the programmer intended gets murky, and how additional information outside the buggy area needs to be brought to bear. We see again that for the programs in Figure 7 the bug categorization and bug frequencies change depending on what decision is made about the programmer's intention.

Finally, the fact that the programs we have shown are *novices'* programs is really irrelevant to the point in question: the problem is that the intention of the programmer effects the bug categorization and the bug count. Quite reasonably, we would not expect a professional programmer to mistake an IF for a WHILE. The observation that we would not expect this particular confusion would in fact aid us in inferring the intention --- it would not, we believe, simply make the problem go away. In fact, we might well see buggy code such as Figure 4, Figure 7 from a professional programmer.

```

a
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
  BEGIN
    IF RAINFALL < 0 THEN
      WRITELN('BAD INPUT TRY AGAIN');
    TOTAL = TOTAL + RAINFALL
    DAYS = DAYS + 1
    READ(RAINFALL)
  END
END

```

```

b
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
  BEGIN
    IF RAINFALL < 0 THEN
      BEGIN
        WRITELN('BAD INPUT TRY AGAIN');
        READ(RAINFALL);
      END
    TOTAL = TOTAL + RAINFALL
    DAYS = DAYS + 1
    READ(RAINFALL)
  END
END

```

```

c
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
  BEGIN
    WHILE RAINFALL < 0 DO
      WRITELN('BAD INPUT TRY AGAIN');
    TOTAL = TOTAL + RAINFALL
    DAYS = DAYS + 1
    READ(RAINFALL)
  END
END

```

Figure 7: Clouding the Waters: Additional Buggy Programs

3. Methods for Specifying the Intention of a Program

In the above section, the basis for describing bugs was the difference between a program and the programming plans that specified a correct program. There are other methods of specifying the intention of a program:

- I/O Behavior
- Programming Plans
- Corrected Version of the Buggy Program
- Program Description Language (PDL)

In what follows we will examine each of these in turn, and explore their good points and the bad points with respect to using a method as a basis for developing bug difference descriptions.

I/O BEHAVIOR

An I/O specification for the problem in Figure 1 would be quite close to the problem statement itself. The obvious problem with this method is its vagueness with respect to the code: many different code fragments can misbehave in the same manner (e.g., there are many, many ways to generating an infinite loop -- but the I/O result is the same in all cases). One needs to be able to make finer-grain distinctions than are facilitated by a comparison of the code to simply I/O

specifications.

PROGRAMMING PLANS

The major problem with this method is the need to guess what plan the programmer intended to implement. However, once the decision is made, then describing the bug as a difference between the plan and the code is relatively easy. One method of coping with the plan decision problem is interviews with the original programmers; this technique has been used to "validate" change report data in several software monitoring projects (e.g., [12]). Unfortunately, in a class of 200 students writing code at different terminals, interviews with subjects is a bit more difficult.

The major benefit derived from building a bug description using this method is an accurate reporting of the *cause* of the bug. That is, clearly the goal of a bug taxonomy in which one captures bug type and bug frequency is the ability to pinpoint the sources of the bugs: one would like to know which bugs came from misunderstandings of the specifications document and which bugs arose from coding errors, etc. For example, in the previous section if we assumed that the programmer intended to implement a *skip-guard plan* then we would say that there were a number of coding level bugs (e.g., WHILE instead of IF, missing ELSE, spurious READ). However, if we assume that the programmer intended to implement an *embedded filter loop plan*, then the source of the bug may be a problem of specification interpretation: the programmer may not have thought that someone would ever input the sentinel value after inputting an illegal (negative) value. Thus he felt no need to guard subsequent computation. (An interview with the programmer would be particularly useful in this specific case.) Thus, bug categorization and *bug origin* is directly influenced by the choice of underlying plan structure in the buggy program.

CORRECTED VERSION OF THE BUGGY PROGRAM

The typical method of describing a bug is to compare the original buggy program with the corrected version of that program (e.g., [12, 7, 1]). While there is no guessing as to the intention of the original programmer, we see 2 basic problems with this approach:

- *The choice of the particular corrected program used as the measure is relatively arbitrary.* That is, there are few hard guidelines for making changes to code. Thus, different programmers could well take the same buggy program and correct it in different ways. This would result in two different bug descriptions — an intuitively unsatisfactory situation. Moreover, different bug descriptions could lead to different conclusions as to the origins of the bugs, which, after all, is the the point of doing the bug categorization in the first place. For example, if the buggy program in Figure 4 were corrected by implementing a *skip-guard plan*, then the difference between the buggy program and the corrected program would result in a bug description containing 3 coding level bugs. On the other hand, if the program is corrected by putting in a guard around the subsequent computation to protect against a sentinel value, then the bug description would only contain 1 bug, a missing conditional

(guard plan) — which may or may not be a coding level bug (as discussed above). While we might prefer the programmer to make the latter change, there is no way to guarantee this situation.

Interviewing the *original programmer* might shed some light on his intentions — and guide the subsequent bug analysis or even bug correction. However, this additional, programmer-supplied, information goes beyond the corrected program — and approaches a bug description based on *the programmers original plan*. While we have some methodological reservations about using interviews collected after the fact,⁵ the main issue is that information gotten from the interview is of a different sort than the information gotten from the corrected program — where the former information is much more akin to the programming plans described above.

- What is actually counted can be quite problematic. For example, if we correct the buggy program in Figure 7c by adding the missing ELSE, we also need to add a BEGIN-END block around the running total update and the counter update. Should we count this as 1 bug or 2 bugs? It seems unfair to count the BEGIN-END block against the programmer, since this change is required by the "real" change. On the other hand, however, in the next section we will show programs in which the "real" bug is a missing BEGIN-END block. Thus, it is not inconceivable that a programmer could add the ELSE in Figure 7c, but forget to put in the now necessary BEGIN-END block. What one counts is a tricky issue.

The upshot of these two problems with categorizing and counting bugs based on a corrected version of the program was suggested above: one is less confident of the origins of the bugs, and thus is less confident about percentages of bugs with those origins. Depending on the particular corrected solution and the particular choice of counting scheme, one could paint a picture of a program that contained many more coding level errors, say, than specification-based errors. The worst part of this situation is that we would not have a good way of knowing how right or wrong this analysis was — since we don't know how the bug categories and counts would have turned out if a different corrected version were used as the basis for difference descriptions.

PROGRAM DESCRIPTION LANGUAGE (PDL)

PDL's come in all flavors; some are very close to the code, while others are more high level, and closer to the plan level description. The former PDL would suffer from the same problems as using a corrected version as the standard. The latter type of PDL would suffer from the problems associated with using the programming plans as the standard.

⁵The problems with using interview data has received significant attention in psychology. For example, Ericsson and Simon [4] have argued that one can reliably only use verbal information given by the subject as the subject is doing the task. They argue that such a concurrent verbal report is effectively an on-line dump from short-term memory. In contrast, a report after the fact could be a story about what the subject thought he was thinking, and that significant distortions can occur in this type of situation. While one might arguably feel that the Ericsson and Simon position is a bit extreme, nonetheless, it seems only prudent to exercise care in interpreting interview data.

4. An Extended Example

Let us now consider an actual example from the on-line protocol data. In Figure 8 we depict the problem the students were trying to solve; in Figure 9 the program on the left is a buggy program generated by a student in our study. If we take a "local view" of the bugs in this program, we can generate a corrected version as shown in Figure 9 (right hand side). Notice that if we do a difference description between the corrected and the buggy versions we can come up with 8 changes:

- The rainyday counter, COUNT1, will be always be updated; in order to correct for the times when a negative rainfall is input, we need to decrement COUNT1. Thus, [1] *added a begin-end block after (NUM < 0) test, and [2] added a decrement of the rainyday counter.*
- COUNT2 must be made to contain the number of rainy (not just valid) days. COUNT2 keeps track of the non-rainy valid days in the loop. Thus, we need to subtract the non-rainy days (COUNT2) from the total valid days (COUNT1) in order to get the number of rainy days: [3] *changed addition of COUNT1 and COUNT2 to subtraction of COUNT2 from COUNT1.*
- The guard on the average calculation is incorrect. Thus, [4] *changed guard on average calculation to COUNT1.*
- The divisor in the average calculation should be the valid day counter, COUNT1, not the valid, but non-rainy day counter, COUNT2. Thus, [5] *changed COUNT2 to COUNT1 in the divisor of the average calculation.*
- If there is no valid input the program should neither calculate the average, nor should the program print it out -- as well as not printing out the maximum. Thus, [6] *added a begin-end block after division guard around average calculation and output statements.*
- The WRITELNs give a message about what should be output; in order to make the message agree with the actual output, the variables need to be changed: [7] *the valid day counter needs to be COUNT1, while the [8] rainy day counter needs to COUNT2.*

Given the number of changes that need to be made to the counters (COUNT1 and COUNT2), it would appear that the student has some confusion over the roles of the two counters.

The Noah Problem: Noah needs to keep track of the rainfall in the New Haven area to determine when to launch his ark. Write a program which he can use to do this. Your program should read the rainfall for each day, stopping when Noah types "99999", which is not a data value, but a sentinel indicating the end of input. If the user types in a negative value the program should reject it, since negative rainfall is not possible. Your program should print out the number of valid days typed in, the number of rainy days, the average rainfall per day over the period, and the maximum amount of rainfall that fell on any one day.

Figure 8: The Noah Problem: A First Looping Problem

However, consider now a different corrected version of this buggy program as depicted in Figure 10. A difference description between the buggy version and the corrected version yields the following set of bugs:

- We can make COUNT1 only keep track of the rainy days; this is consistent with code

BUGGY EXAMPLE

```

BEGIN
WRITELN ('PLEASE! INPUT AMOUNT OF RAINFALL')
READLN
READ(NUM)
COUNT1 = 0
COUNT2 = 0
SUM = 0
HIGHNUM = 0
WHILE (NUM <> SENTINAL) DO
  BEGIN
    IF (NUM > 0)
    THEN
      SUM = SUM + NUM
      COUNT1 = COUNT1 + 1
      IF (NUM > HIGHNUM)
      THEN
        HIGHNUM = NUM
    IF (NUM = 0)
    THEN
      COUNT2 = COUNT2 + 1
    IF (NUM < 0)
    THEN
      WRITELN ('ILLEGAL INPUT INPUT NEW VALUE')
      READLN
      READ(NUM)
    END
  COUNT2 = COUNT2 + COUNT1
  IF (NUM > 0)
  THEN
    TOTAL = SUM/COUNT2
    WRITELN ('AVERAGE RAINFALL WAS ' TOTAL ' INCHES PER DAY')
    WRITELN ('HIGHEST RAINFALL WAS ' HIGHNUM ' INCHES')
  WRITELN (COUNT2 ' VALID DAYS WERE ENTERED')
  WRITELN (COUNT1 ' RAINY DAYS IN THIS PERIOD ')
  END

```

CORRECTED VERSION

```

BEGIN
WRITELN ('PLEASE! INPUT AMOUNT OF RAINFALL')
READLN
READ(NUM)
COUNT1 = 0
COUNT2 = 0
SUM = 0
HIGHNUM = 0
WHILE (NUM <> SENTINAL) DO
  BEGIN
    IF (NUM > 0)
    THEN
      SUM = SUM + NUM
      COUNT1 = COUNT1 + 1
      IF (NUM > HIGHNUM)
      THEN
        HIGHNUM = NUM
    IF (NUM = 0)
    THEN
      COUNT2 = COUNT2 + 1
    IF (NUM < 0)
    THEN
      begin (* add this line *)
      COUNT1 = COUNT1 - 1; (* add this line *)
      WRITELN ('ILLEGAL INPUT INPUT NEW VALUE')
      end; (* add this line *)
      READLN
      READ(NUM)
    END
    COUNT2 = COUNT1 - COUNT2; (* changed this line *)
    IF (COUNT1 > 0) (* changed this line *)
    THEN
      begin (* add this line *)
      TOTAL = SUM/COUNT1, (* changed this line *)
      WRITELN ('AVERAGE RAINFALL WAS ' TOTAL ' INCHES PER DAY')
      WRITELN ('HIGHEST RAINFALL WAS ' HIGHNUM ' INCHES')
      end; (* add this line *)
    WRITELN(COUNT1 ' VALID DAYS WERE ENTERED') (* changed this line *)
    WRITELN(COUNT2 ' RAINY DAYS IN THIS PERIOD ') (* changed this line *)
  END

```

- [1] added a begin-end block after (NUM < 0) test, and [2] added a decrement of the rainyday counter
- [3] changed addition of COUNT1 and COUNT2 to subtraction of COUNT2 from COUNT1
- [4] changed guard on average calculation to COUNT1
- [5] changed COUNT2 to COUNT1 in the divisor of the average calculation
- [6] added a begin-end block after division guard around average calculation and output statements
- [7] the valid day counter needs to be COUNT1 while the [8] rainy day counter needs to COUNT2

Figure 9: A Buggy Program and a Corrected Version

already in the program: the line that adds COUNT2 and COUNT1 now makes sense --- COUNT2 now keeps track of the valid days, and the divisor in the average calculation suggests that COUNT2 should be the valid day counter. In order to make COUNT1 perform in this manner, we need to [1] *add a begin-end pair around all computation after NUM > 0 test, up to the NUM = 0 test.*

- If there is no valid input the program should neither calculate the average, nor should the program print it out --- as well as not printing out the maximum. Thus, we need to [2] *add a begin-end block after division guard around average calculation and output statements.*
- The guard on the average calculation is incorrect. Thus, [3] *changed guard on average calculation to COUNT1.*

Which description should we choose? And why? Notice that neither of the corrected versions were that unreasonable. However, it would seem to us that one should choose the second bug description over the first. The basis for that decision is the hypothesized plan structure underlying the buggy version: it appears to us that the student was trying to structure the actions in the main loop in terms of cases. For example, the program explicitly tested for $NUM > 0$, $NUM = 0$, and $NUM < 0$ and took the appropriate actions --- almost. In order to make the case structure work, the code following the $NUM > 0$ up to the $NUM = 0$ test should be grouped together. While one cannot put too much faith in the indentation of a novice's program,⁶ it appears that the indentation supports this analysis. Thus, what is missing from the main loop is a *begin-end* pair surrounding the code between the $NUM > 0$ test and the $NUM = 0$ test. On this analysis, the student does not have a misunderstanding surrounding the two counters, but rather has a coding level misunderstanding about how to block code together. Moreover, this same misunderstanding can explain the lack of a *begin-end* pair surrounding the average calculation in the next two write statements. The reduced bug count in the second description follows directly from this analysis: in effect there are only 3 bugs in this program, 2 of which have the same underlying origin.

This example illustrates a point made earlier: *the bug categorization and bug count follow from an understanding of the program that is provided by the hypothesized plan structure of the program.* That is, to understand a buggy program, one must make inferences about what plan structure the programmer intended to implement; the program only "makes sense" in terms of these plan descriptions.

⁶We have observed in the on-line protocols that the physical layout of a student's program suffers as the student makes changes to his program in the process of debugging it.

BUGGY EXAMPLE

```

BEGIN
  WRITELN ('PLEASE! INPUT AMOUNT OF RAINFALL').
  READLN
  READ(NUM)
  COUNT1 = 0
  COUNT2 = 0
  SUM = 0
  HIGHNUM = 0
  WHILE (NUM <> SENTINAL) DO
    BEGIN
      IF (NUM > 0)
      THEN
        SUM = SUM + NUM
        COUNT1 = COUNT1 + 1
        IF (NUM > HIGHNUM)
        THEN
          HIGHNUM = NUM
      IF (NUM = 0)
      THEN
        COUNT2 = COUNT2 + 1
      IF (NUM < 0)
      THEN
        WRITELN ('ILLEGAL INPUT INPUT NEW VALUE')
      READLN
      READ(NUM)
    END
    COUNT2 = COUNT2 + COUNT1
    IF (NUM > 0)
    THEN
      TOTAL = SUM/COUNT2
      WRITELN ('AVERAGE RAINFALL WAS ' TOTAL ' INCHES PER DAY')
      WRITELN ('HIGHEST RAINFALL WAS ' HIGHNUM ' INCHES')
    WRITELN (COUNT2 ' VALID DAYS WERE ENTERED')
    WRITELN (COUNT1 ' RAINY DAYS IN THIS PERIOD ')
  END

```

ANOTHER CORRECTED VERSION

```

BEGIN
  WRITELN ('PLEASE! INPUT AMOUNT OF RAINFALL').
  READLN
  READ(NUM)
  COUNT1 = 0
  COUNT2 = 0
  SUM = 0
  HIGHNUM = 0
  WHILE (NUM <> SENTINAL) DO
    BEGIN
      IF (NUM > 0)
      THEN
        begin
          SUM = SUM + NUM (* add this line *)
          COUNT1 = COUNT1 + 1
          IF (NUM > HIGHNUM)
          THEN
            HIGHNUM = NUM
          end; (* add this line *)
      IF (NUM = 0)
      THEN
        COUNT2 = COUNT2 + 1
      IF (NUM < 0)
      THEN
        WRITELN ('ILLEGAL INPUT INPUT NEW VALUE')
      READLN
      READ(NUM)
    END
    COUNT2 = COUNT2 + COUNT1
    IF (COUNT2 > 0) (* changed this line *)
    THEN
      begin
        TOTAL = SUM/COUNT2 (* add this line *)
        WRITELN ('AVERAGE RAINFALL WAS ' TOTAL ' INCHES PER DAY')
        WRITELN ('HIGHEST RAINFALL WAS ' HIGHNUM ' INCHES ')
      end; (* add this line *)
    WRITELN (COUNT2 ' VALID DAYS WERE ENTERED')
    WRITELN (COUNT1 ' RAINY DAYS IN THIS PERIOD ')
  END

```

- [1] add a begin-end pair around all computation after NUM > 0 test up to the NUM = 0 test
- [2] add a begin-end block after division guard around average calculation and output statements
- [3] changed guard on average calculation to COUNT1

Figure 10: A Buggy Program and an Alternative Corrected Version

5. Concluding Remarks

We have argued that a bug description is a difference description between the realization and the intention specification. We have presented a number of techniques for specifying the intention and have pointed out the problems associated with each type of specification in developing an accurate picture of bug types and bug frequency. While no technique is without its problems, we have argued that the understanding provided by a plan analysis of the buggy program stands a better chance, as compared to the other techniques, of providing a more accurate categorization and count of the bugs -- and thus a more accurate reflection of the origins of the bugs.

References

1. Basili, V., Perricone, B. Software Errors and Complexity: An Empirical Investigation. Tech. Rept. TR-1195, University of Maryland, Dept. of Computer Science, 1982.
2. Bonar, J. Understanding the Novice Programmer. Dissertation, in preparation.
3. Ehrlich, K., Soloway, E. An Empirical Investigation of the Tacit Plan Knowledge in Programming. in *Human Factors in Computer Systems*, J. Thomas and M.L. Schneider (Eds.), Ablex Inc., in press.
4. Ericsson, A. and Simon, H. "Verbal reports as data." *Psychological Review* 87 (1980), 215-251.
5. Johnson, L., Draper, S., Soloway, E. The Nature of Bugs in Novices' Pascal Programs. in preparation
6. Miller, L. A. "Natural Language Programming: Styles, Strategies, and Contrasts." *IBM Systems Journal* 20 (1981), 184-215.
7. Ostrand, T., Weyuker, E. Collecting and Categorizing Software Error Data in an Industrial Environment. Tech. Rept. 47, New York University, Dept. of Computer Science, 1982.
8. Rich, C. Inspection Methods in Programming. Tech. Rept. AI-TR-604, MIT AI Lab, 1981.
9. Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J. What Do Novices Know About Programming? In A. Badre, B. Shneiderman, Ed., *Directions in Human-Computer Interactions*, Ablex, Inc., 1982.
10. Soloway, E., Bonar, J., Ehrlich, K. . Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, in press.
11. Soloway, E., Rubin, E., Woolf, B., Bonar, J., Johnson, L. MENO-II: An Intelligent Programming Tutor. *Journal of Computer-Based Instruction*, to appear.
12. Weiss, D. Evaluating Software Development By Analysis of Change Data. Tech. Rept. TR-1120, University of Maryland, Dept. of Computer Science, 1981.

- OFFICIAL DISTRIBUTION LIST -

Army		Private Sector	
Technical Director U S Army Research Institute for the Behavioral and Social Sciences 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Michael Genesereth Department of Computer Science Stanford University Stanford, California 94305	1 copy
Mr. James Baker Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Dedre Gentner Bolt Beranek & Newman 10 Moulton Street Cambridge, Massachusetts 02138	1 copy
Dr. Beatrice J. Farr U S Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Robert Glaser Learning Research & Development Center University of Pittsburgh 3939 O'Hara Street Pittsburgh, Pennsylvania 15260	1 copy
Dr. Milton S. Katz Williams Technical Area U S Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Joseph Goguen SRI International 333 Ravenswood Avenue Menlo Park, California 94025	1 copy
Dr. Marshall Marva U S Army Research Institute for the Behavioral & Social Sciences 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Bert Green Johns Hopkins University Department of Psychology Charles & 34th Street Baltimore, Maryland 21218	1 copy
Dr. Harold F. O'Neill, Jr. Director, Training Research Lab Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. James G. Greeno LRDC University of Pittsburgh 3939 O'Hara Street Pittsburgh, Pennsylvania 15213	1 copy
Commander, US Army Research Institute for the Behavioral & Social Sciences Attn: PERI-BR (Dr. Judith Orasanu) 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Barbara Hayes-Roth Department of Computer Science Stanford University Stanford, California 95305	1 copy
Joseph Psotka, Ph.D. Attn: PERI-IC Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Frederick Hayes-Roth Teknowledge 525 University Avenue Palo Alto, California 94301	1 copy
Dr. Robert Sasmor U S Army Research Institute for the Behavioral and Social Sciences 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Glen Greenwald, Ed Human Intelligence Newsletter P O Box 1163 Birmingham, Michigan 48012	1 copy
Dr. Robert Wisher Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333	1 copy	Dr. Earl Hunt Department of Psychology University of Washington Seattle, Washington 98105	1 copy
		Dr. Marcel Just Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy

Air Force

U S Air Force Office of Scientific Research Life Sciences Directorate, NL Boiling Air Force Base Washington, DC 20332	1 copy	Dr. David Kieras Department of Psychology University of Arizona Tucson, Arizona 85721	1 copy
Dr. Earl A. Allais HQ AFHRL (AFSC) Brooks AFB, Texas 78235	1 copy	Dr. Walter Kintsch Department of Psychology University of Colorado Boulder, Colorado 80302	1 copy
Bryan Dailman AFHRL/LRT Lowry AFB, Colorado 80230	1 copy	Dr. Stephen Kosslyn Department of Psychology The Johns Hopkins University Baltimore, Maryland 21218	1 copy
Dr. Genevieve Haddad Program Manager Life Sciences Directorate AFOSR Boiling AFB, DC 20332	1 copy	Dr. Pat Langley The Robotics Institute Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy
Dr. John Tangney AFOSR/NL Boiling AFB, DC 20332	1 copy	Dr. Jill Larkin Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy
Dr. Joseph Yasutake AFHRL/LRT Lowry AFB, Colorado 80230	1 copy	Dr. Alan Lesgold Learning R&D Center University of Pittsburgh 3939 O'Hara Street Pittsburgh, Pennsylvania 15213	1 copy
Marine Corps		Dr. Jim Levin University of California at San Diego Laboratory for Comparative Human Cognition - 0003A La Jolla, California 92093	1 copy
H. William Greenup Education Advisor (E031) Education Center, MCDEC Quantico, Virginia 22134	1 copy	Dr. Michael Levine Department of Educational Psychology 210 Education Bldg University of Illinois Champaign, Illinois 61801	1 copy
Special Assistant for Marine Corps Matters Code 150M Office of Naval Research 800 N. Quincy Street Arlington, Virginia 22217	1 copy	Dr. Marcia Linn University of California Director, Adolescent Reasoning Project Berkeley, California 94720	1 copy
Dr. A. L. Slafkosh Scientific Advisor (Code RD-1) HQ, U S Marine Corps Washington, DC 20380	1 copy	Dr. Jay McClelland Department of Psychology MIT Cambridge, Massachusetts 02139	1 copy
Department of Defense		Dr. James R. Miller Computer Thought Corporation 1721 West Plano Highway Plano, Texas 75075	1 copy
Defense Technical Information Center Cameron Station, Bldg 5 Alexandria, Virginia 22314 Attn: TC	12 copies	Dr. Mark Miller Computer Thought Corporation 1721 West Plano Highway Plano, Texas 75075	1 copy
Military Assistant for Training and Personnel Technology Office of the Under Secretary of Defense for Research & Engineering Room, 3D129, The Pentagon Washington, DC 20301	1 copy		
Major Jack Thorpe DARPA 1400 Wilson Blvd Arlington, Virginia 22209	1 copy		

Navy		Dr. Tom Morse	1 copy
Robert Aiders	1 copy	Kerns PARC	
Code 8711		3333 Coyote Hill Road	
Human Factors Laboratory		Palo Alto, California 94304	
NAVTRAEQUIPCEN		Dr. Allen Nuevo	1 copy
Orlando, Florida 32813		Behavioral Technology Laboratories	
Code 8711	1 copy	1045 Elena Avenue, Fourth Floor	
Attn: Arthur S. Blaines		Redondo Beach, California 90277	
Naval Training Equipment Center		Dr. Donald Norman	1 copy
Orlando, Florida 32813		Cognitive Science, C-015	
Liaison Scientist	1 copy	Univ. of California, San Diego	
Office of Naval Research		La Jolla, California 92093	
Branch Office, London		Dr. Jesse Orlesky	1 copy
Box 39		Institute for Defense Analyses	
FPD New York, New York 09510		1801 N. Beauregard Street	
Dr. Richard Cantone	1 copy	Alexandria, Virginia 22311	
Navy Research Laboratory		Professor Seymour Papert	1 copy
Code 7510		26C-109	
Washington, DC 20375		MIT	
Chief of Naval Education and Training	1 copy	Cambridge, Massachusetts 02139	
Liaison Office		Dr. Nancy Pennington	1 copy
Air Force Human Resource Laboratory		University of Chicago	
Operations Training Division		Graduate School of Business	
WILLIAMS AFB, Arizona 85224		1101 E. 58th Street	
Dr. Stanley Collier	1 copy	Chicago, Illinois 60637	
Office of Naval Technology		Dr. Richard A. Pollat	1 copy
800 N. Quincy Street		Director, Special Projects	
Arlington, Virginia 22217		MECC	
CDR Mike Curran	1 copy	2354 Hidden Valley Lane	
Office of Naval Research		Stillwater, Minnesota 55082	
800 N. Quincy Street		Dr. Peter Polson	1 copy
Code 270		Department of Psychology	
Arlington, Virginia 22217		University of Colorado	
Dr. John Ford	1 copy	Boulder, Colorado 80309	
Navy Personnel R&D Center		Dr. Fred Reif	1 copy
San Diego, California 92152		Physics Department	
Dr. Jude Franklin	1 copy	University of California	
Code 7510		Berkeley, California 94720	
Navy Research Laboratory		Dr. Lauren Resnick	1 copy
Washington, DC 20375		LRDC	
Dr. Mike Gaylor	1 copy	University of Pittsburgh	
Navy Research Laboratory		3938 O'Hara Street	
Code 7510		Pittsburgh, Pennsylvania 15213	
Washington, DC 20375		Mary S. Riley	1 copy
Dr. Jim Hoffman	1 copy	Program in Cognitive Science	
Code 14		Center for Human Information Processing	
Navy Personnel R&D Center		University of California, San Diego	
San Diego, California 92152		La Jolla, California 92093	
Dr. Ed Hutchins	1 copy	Dr. Andrew Rose	1 copy
Navy Personnel R&D Center		American Institution for Research	
San Diego, California 92152		1086 Thomas Jefferson Street, NW	
		Washington, DC 20007	

Dr. Norman J. Kerr Chief of Naval Technical Training Naval Air Station Memphis (75) Hillington, Tennessee 38054	1 copy	Dr. Ernst Z. Rothkopf Bell Laboratories Murray Hill, New Jersey 07974	1 copy
Dr. James Lester ONR Detachment 495 Summer Street Boston, Massachusetts 02210	1 copy	Dr. William B. Rouse Georgia Institute of Technology School of Industrial & Systems Engineering Atlanta, Georgia 30332	1 copy
Dr. William L. Maloy (02) Chief of Naval Education and Training Naval Air Station Pensacola, Florida 32508	1 copy	Dr. David Rumelhart Center for Human Information Processing University of California, San Diego La Jolla, California 92093	1 copy
Dr. Joe McLachlan Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Michael J. Samet Perceptronics, Inc. 6271 Varrel Avenue Woodland Hills, California 91364	1 copy
Dr. William Montague NFRDC Code 13 San Diego, California 92152	1 copy	Dr. Roger Schank Yale University Department of Computer Science P.O. Box 2150 New Haven, Connecticut 06520	1 copy
Library Code P201L Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Walter Schneider Psychology Department 603 E. Daniel Champaign, Illinois 61820	1 copy
Technical Director Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Alan Schoenfeld Mathematics and Education The University of Rochester Rochester, New York 14627	1 copy
Commanding Officer Naval Research Laboratory Code 2627 Washington, DC 20390	6 copies	Mr. Colin Sheppard Applied Psychology Unit Admiralty Marine Technology Est. Teddington, Middlesex United Kingdom	1 copy
Office of Naval Research Code 433 806 N. Quincy Street Arlington, Virginia 22217	1 copy	Dr. H. Wallace Simaiko Program Director Manpower Research and Advisory Service Smithsonian Institution 801 North Pitt Street Alexandria, Virginia 22314	1 copy
Personnel & Training Research Group Code 442PT Office of Naval Research Arlington, Virginia 22217	6 copies	Dr. Edward E. Smith Bolt Beranek & Newman 50 Mounton Street Cambridge, Massachusetts 02138	1 copy
Office of the Chief of Naval Operations Research Development & Studies Branch OP 115 Washington, DC 20350	1 copy	Dr. Richard Snow School of Education Stanford University Stanford, California 94305	1 copy
LT Frank C. Petto, MSC, USN (Ph D) CNET (N-432) NAS Pensacola, Florida 32508	1 copy	Dr. Kathryn I. Spoeck Psychology Department Brown University Providence, Rhode Island 02912	1 copy
Dr. Gary Poock Operations Research Development Code 58PM Naval Postgraduate School Monterey, California 93940	1 copy		

Dr. Gil Ricard Code M711 NTEC Orlando, Florida 32813	1 copy	Dr. Robert Sternberg Department of Psychology Yale University Box 11A, Yale Station New Haven, Connecticut 06520	1 copy
Dr. North Scanland CNET (N-5) NAS, Pensacola, Florida 32508	1 copy	Dr. Albert Stevens Bolt Beranek & Neuman 10 Moulton Street Cambridge, Massachusetts 02238	1 copy
Dr. Robert G. Smith Office of Chief of Naval Operations OP-987H Washington, DC 20350	1 copy	David E. Stone, Ph.D. Hazeltime Corporation 7680 Old Springhouse Road McLean, Virginia 22102	1 copy
Dr. Alfred F. Smode, Director Training Analysis & Evaluation Group Department of the Navy Orlando, Florida 32813	1 copy	Dr. Patrick Suppes Institute for Mathematical Studies in the Social Sciences Stanford University Stanford, California 94305	1 copy
Dr. Richard Sorensen Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Kikumi Tatsuka Computer Based Education Research Lab 252 Engineering Research Laboratory Urbana, Illinois 61801	1 copy
Dr. Frederick Steinheiser CNO - OP115 Navy Annex Arlington, Virginia 20370	1 copy	Dr. Maurice Tatsuka 220 Education Bldg. 1310 S. Sixth Street Champaign, Illinois 61820	1 copy
Roger Weissinger-Baylon Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1 copy	Dr. Perry W. Thorndyke Perceptronics, Inc. 545 Middlefield Road, Suite 140 Menlo Park, California 94025	1 copy
Mr. John H. Wolfe Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Douglas Towne University of So. California Behavioral Technology Labs 1845 S. Elena Avenue Redondo Beach, California 90277	1 copy
Dr. Wallace Bullock III Navy Personnel R&D Center San Diego, California 92152	1 copy	Dr. Kurt Van Lehn Xerox PARC 3333 Coyote Hill Road Palo Alto, California 94304	1 copy
Private Sector		Dr. Keith T. Wescoart Perceptronics, Inc. 545 Middlefield Road, Suite 140 Menlo Park, California 94025	1 copy
Dr. John R. Anderson Department of Psychology Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	1 copy	Dr. John Annett Department of Psychology University of Warwick Coventry CV4 7AJ ENGLAND	1 copy
Dr. Michael Atwood ITT - Programming 1000 Gronoques Lane Stratford, Connecticut 06497	1 copy	William B. Whitten Bell Laboratories 2D-610 Holmdel, New Jersey 07733	1 copy
Dr. Alan Baddeley Medical Research Council Applied Psychology Unit 15 Chaucer Road Cambridge CB2 2EF ENGLAND	1 copy	Dr. Mike Williams Xerox PARC 3333 Coyote Hill Road Palo Alto, California 94304	1 copy

			Civilian Agencies	
Dr. Patricia Baggett	1 copy	Dr. Patricia A. Butler	1 copy	
Department of Psychology		NIE-ORN Bldg. Stop 67		
University of Colorado		1200 19th Street NW		
Boulder, Colorado 80309		Washington, DC 20208		
Ms. Carole A. Bagley	1 copy	Dr. Susan Chipman	1 copy	
Minnesota Educational Computing Consortium		Learning and Development		
2354 Hidden Valley Lane		National Institute of Education		
Stillwater, Minnesota 55082		1200 19th Street NW		
		Washington, DC 20208		
Dr. Jonathan Baron	1 copy	Edward Esty	1 copy	
80 Glenn Avenue		Department of Education, OERI		
Berwyn, Pennsylvania 19312		MS 40		
		1200 19th Street, NW		
Mr. Avron Barr	1 copy	Washington, DC 20208		
Department of Computer Science		Edward J. Fuentes	1 copy	
Stanford University		Department of Education		
Stanford, California 94305		1200 19th Street, NW		
		Washington, DC 20208		
Dr. John Black	1 copy	TABE, TAK	1 copy	
Yale University		National Institute of Education		
Box 114 Yale Station		1200 19th Street, NW		
New Haven, Connecticut 06520		Washington, DC 20208		
Dr. John S. Brown	1 copy	Dr. John Mays	1 copy	
XEROX Palo Alto Research Center		National Institute of Education		
3533 Coyote Road		1200 19th Street, NW		
Palo Alto, California 94304		Washington, DC 20208		
Dr. Bruce Buchanan	1 copy	Dr. Arthur Melmed	1 copy	
Department of Computer Science		724 Brown		
Stanford University		U. S. Dept. of Education		
Stanford, California 94305		Washington, DC 20208		
Dr. Jaime Cartorelli	1 copy	Dr. Andrew R. Molnar	1 copy	
Department of Psychology		Office of Scientific and Engineering		
Carnegie-Mellon University		Personnel and Education		
Pittsburgh, Pennsylvania 15213		National Science Foundation		
		Washington, DC 20550		
Dr. Pat Carpenter	1 copy	Everett Palmer	1 copy	
Department of Psychology		Research Scientist		
Carnegie-Mellon University		Mail Stop 239-3		
Pittsburgh, Pennsylvania 15213		NASA Ames Research Center		
		Moffett Field, California 94035		
Dr. William Chase	1 copy	Dr. Mary Stoddard	1 copy	
Department of Psychology		C 10, Mail Stop 8296		
Carnegie-Mellon University		Los Alamos National Laboratories		
Pittsburgh, Pennsylvania 15213		Los Alamos, New Mexico 87545		
Dr. Micheline Chi	1 copy	Chief, Psychological Research Branch	1 copy	
Learning R & D Center		U. S. Coast Guard (G-P-1/2/TP42)		
University of Pittsburgh		Washington, DC 20583		
3839 O'Hara Street				
Pittsburgh, Pennsylvania 15213				

Dr. William Clancey
Department of Computer Science
Stanford University
Stanford, California 94306

1 copy

Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

1 copy

ERIC Facility-Acquisitions
4833 Rugby Avenue
Bethesda, Maryland 20014

1 copy

Mr. Wallace Feurzeig
Department of Educational Technology
Bolt Beranek and Newman
10 Moulton Street
Cambridge, Massachusetts 02238

1 copy

Dr. Dexter Fletcher
WICAT Research Institute
1875 S. State Street
Orem, Utah 84403

1 copy

Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, Massachusetts 02138

1 copy

Dr. Frank Withrow
U. S. Office of Education
400 Maryland Avenue SW
Washington, DC 20202

1 copy

Dr. Joseph L. Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

1 copy